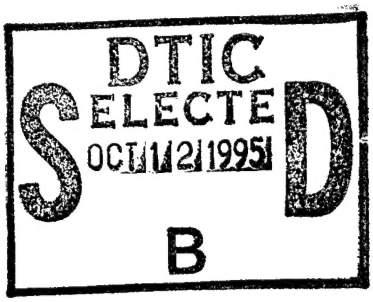
ARMY RESEARCH LABORATORY

A Slicing Method for Semantics-Based Change-Merging of Software Prototypes

CPT David A. Dampier
U.S. ARMY RESEARCH LABORATORY

Valdis Berzins
Luqi
Mantak Shing
Daniel R. Dolk
Craig W. Rasmussen
NAVAL POSTGRADUATE SCHOOL



ARL-TR-840 August 1995

19951011 076

NOTICES

Destroy this report when it is no longer needed. DO NOT return it to the originator.

Additional copies of this report may be obtained from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Road, Springfield, VA 22161.

The findings of this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

The use of trade names or manufacturers' names in this report does not constitute indorsement of any commercial product.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 1995	Final, Jan 93 - Sep 94

4. TITLE AND SUBTITLE

A Slicing Method For Semantics-Based Change-Merging of Software Prototypes

5. FUNDING NUMBERS

N/A

6. AUTHOR(S)

CPT David A. Dampier, Valdis Berzins, Luqi, Mantak Shing, Daniel R. Dolk, and Craig R. Rasmussen

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

U.S. Army Research Laboratory
ATTN: AMSRL-SC-IS
115 O'Keefe Bldg., GIT
Atlanta, GA 30332-0800

CS, SM and MA Departments
Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

ARL-TR-840

9. SPONSORING/MONITORING AGENCY NAMES(S) AND ADDRESS(ES)

10.SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

This report was presented as a paper at the Computers in Engineering Symposium in Houston, TX, on 30 January 1995.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report outlines a formal method for merging changes in independently developed versions of software prototypes. A useful semantics-based method, which is guaranteed to detect all conflicts, is outlined. Prototype slicing is used to determine the affected parts of each variation and the preserved part of the base in both variations. The affected parts are then combined with the preserved part to complete the merge. Our slicing theorem guarantees that this method produces a prototype that correctly exhibits the significant behavior of each of the input versions, provided the changes do not conflict. Correctness is achieved by comparing the slices of the variation and the merged program with respect to the affected parts of each variation. If the slices are the same, then the result is correct, otherwise a diagnostic message results. Preliminary testing shows that this tool will enhance the ability of the prototype developer to deliver a prototype more quickly by allowing more concurrency in the development effort.

14. SUBJECT TERMS

change-merging, formal methods, software development, semantics-based, prototype

15. NUMBER OF PAGES

21

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

INTENTIONALLY LEFT BLANK.

TABLE OF CONTENTS

INTENTIONALLY LEFT BLANK.

LIST OF FIGURES

INTENTIONALLY LEFT BLANK.

1. INTRODUCTION

During iterative development of software prototypes, different variations are generally developed where each of the versions contains a portion of the desired capability. Because these prototypes can be very large, tools that automatically determine the differences between these versions and produce a new version exhibiting significant behavior from each are desirable. This report describes a change-merging method for the Prototype System Description Language (PSDL) (Luqi, Berzins, and Yeh 1988), a prototype that is semantics-based and guarantees that if a conflict-free result is produced, it is semantically correct. A full definition of this method and the associated tool can be found in Dampier (1994).

2. RAPID PROTOTYPING

Rapid prototyping is an approach to software development that was introduced to overcome the following weaknesses of traditional approaches:

1. fully developed software systems that do not satisfy the customer's needs, or are obsolete upon release
2. no capability for accurately evaluating real-time requirements before the software system has been built

Rapid prototyping overcomes these weaknesses by increasing customer interaction during the requirements engineering phase of development, providing executable specifications that can be evaluated for conformance to real-time requirements, and producing a production software system in a fraction of the time required using traditional methods. Rapid prototyping allows the user to get a better understanding of requirements early in the conceptual design phase of development. It involves the use of software tools to rapidly create concrete executable models of selected aspects of a proposed system to allow the user to view the model and make comments early. The prototype is rapidly reworked and redemonstrated to the user over several iterations until the designer and the user have a precise view of what the system should do. In this approach to rapid prototyping, software systems can be delivered incrementally as parts of the system become fully operational.

Change-merging is an integral part of the rapid prototyping methodology. During prototype development, multiple variations of a large prototype are likely to be developed. This can happen when different development teams are working on different aspects of a system, or when different possible solutions to a problem are explored in different ways. Our change-merging method will allow the combination of these independently developed variations to be done automatically, ensuring that the resultant prototype is semantically correct, with respect to all of the input variations. If the pieces are not compatible with regard to the semantics of the prototype, then our method will identify the parts of the prototype containing the conflicts. This technology encourages the designer to explore different solutions to a problem, and to spread the development workload in a large project without concern for the subsequent integration of these independent efforts.

The earliest work on program merging relied on combining changes made to the text files containing the source code for the program (Silverberg 1992; Tichy 1982). These syntax-based methods proved insufficient to guarantee the correctness of the resultant program. Early semantics-based methods concentrated on higher level domains (Berzins 1986) and simple while programs (Horwitz, Prins, and Reps 1988; Reps and Yang 1988; Yang 1990). This work showed that calculating an exact semantics-based change-merge is not possible in the general case, but useful approximations are possible and feasible.

3. MODEL

PSDL programs are executable specifications that approximate the functionality of a production software system. To describe our method of change-merging as semantics-based, we must first describe the semantics of the language. We chose to model the behavior of a prototype by observing the data flow history over its data streams. A prototype's behavior is represented by sets of possible histories over the streams we call *trace_tuples*. These trace_tuples are composed of sequences of data_tuples called *traces*. Each trace_tuple contains precisely one trace per stream. Since PSDL prototypes are nondeterministic, one trace_tuple does not necessarily reflect the set of possible histories associated with a prototype; thus, we must consider the behavior of a prototype to be the set of all possible trace_tuples over its data streams. Since PSDL prototypes are intended to prototype embedded real-time systems that may never be turned off, this behavior is likely to be of infinite length. The following subsections describe the model starting with traces and building up to the behavior of a prototype and the possibility functions we use to construct the behaviors.

3.1 Traces and Stream Behaviors. A trace is a sequence $\{d_1, d_2, d_3, ...\}$ of possible data tuples written to a data stream. Each d_i contains a data element x, the name o of the operator that wrote x to the stream, the time tw that x was written to the stream, and the time tr that o last read its input streams before producing x. A truncated trace of length k is a sequence containing no more than k data tuples. A stream behavior is a set of possible traces for a stream.

3.2 Trace Tuples and Prototype Behaviors. A trace tuple over a set of streams is a tuple containing one trace for each stream in the set. This trace tuple can be viewed as a tuple of traces or a sequence of incremental trace tuples, according to Dampier's (1994) Theorem 2. It is, in fact, this latter representation that allows us to prove our semantic invariance theorem using induction over the length of the trace tuple. A truncated trace tuple of length k is a tuple of truncated traces of length no more than k.

A prototype's behavior, B, is defined as the set of all possible trace tuples over the streams of the prototype. A truncated behavior of length k, $B \mid k$, is the set of all possible trace tuples for the prototype truncated at length k. Constructing the behavior of a prototype is done inductively by using the prototype's behavior of length k to produce the behavior of length $k + 1$ as follows:

$$\bigcup_{T \in B \mid k} \left[\bigcup_{S \in P(V(Sp(X)))} \left[T \oplus \left(\bigoplus_{v \in S} \left(\bigcup_{\rho(T,v) < tr} \left(\bigcup_{tr < t}^{\Delta} \left(t, fill\left(F_v\left(T_{I_v}, tr \right) \right) \right) \right) \right) \right) \right] \right]$$

This construction uses the prototype's truncated behavior of length k, and for every truncated trace tuple in $B \mid k$, it produces a set of incremental trace tuples that are appended to the end of each of the T's. This construction produces a new set of trace tuples of length no more than $k + 1$. Incremental trace tuples are trace tuples where each trace contains zero or one data tuple.

3.3 Possibility Functions. At the heart of this construction is the possibility function for each operator in the prototype, F_o. To define the possibility function for an operator o, we look at a trace tuple projection of the behavior $\sigma \in B_{I(o)}$ as a sequence of input vectors to o. For every finite prefix of σ applied to o, the result is a set of possible incremental trace tuples over the output streams of o. F_o takes as input a projected trace tuple over the input streams of o and a read time, and produces a set of possible behavior

projections over the output streams of o. The read time is the time at which the last read operation was performed by o on its input streams, and defines which values were read by o to perform this computation.

3.3.1 Example 1: Possibility function for an operator p that implements the function: $y_k = x_k^2$

$$F_p = \{(\{3\},9),(\{3,4\},16),(\{3,4,9\},81),...,(\{3,-4,9,...,x_k\},x_k^2),...\}$$

3.3.2 Example 2: Possibility function for an operator q that implements the state machine:

$$y_k = \sum_{i=1}^{k} x_i$$

$$F_p = \{(\{3\},3),(\{3,-4\},-1),(\{3,-4,9\},8),...,(\{3,4,9,...,x_k\}, \sum_{i=1}^{k-1} x_i + x_k),...\}$$

In Dampier (1994), the Independent Operator Lemma guaranteed us that an operator has the same possibility function regardless of the context in which it is placed; therefore, it can be shown that the inductive construction shown above produces a unique prototype behavior.

3.4 Prototype Slicing. It has been shown that a portion of a program's behavior can be captured by a slice of the program with respect to a single point in the program (Horwitz, Prins, and Reps 1988; Reps 1989; Weiser 1984). We have developed a similar method for isolating a portion of the behavior of a prototype. This section describes our method for taking slices of PSDL prototypes. One of the differences between slicing for PSDL prototypes and slicing for while programs is that PSDL programs are inherently concurrent and nondeterministic. While programs represent individual deterministic sequential processes. This represents a major contribution of this work.

To capture all of the prototype's dependencies using our slicing method, we must enhance the prototype's implementation graph as follows:

3.4.1 Definition 1: PSDL Prototype Dependence Graph: A Prototype Dependence Graph (PDG) for a prototype P is a fully expanded PSDL implementation graph G_P. In the PDG, $G_P = (V, E, C)$, the set of vertices has been augmented with an external vertex, EXT, and the set of edges, E, has been augmented with a timer dependency edge from o_i to o_j, for each pair of vertices o_i, $o_j \in V$, such that the control

with a timer dependency edge from o_i to o_j, for each pair of vertices o_i, $o_j \in V$, such that the control constraints of o_i contain timer operations which affect the state of a timer read by the control constraints of o_j.

A slice of a PSDL prototype is defined in terms of the prototype's dependence graph. It contains the portion of the prototype that affects the history of a set of streams. This is useful in isolating changes made to a base version of a prototype in a modification. If the slices of two versions with respect to the same set of streams are different, then there are significant changes that have been made to one version and not the other.

Informally, a slice is an upstream closure of a set of edges in the graph that includes all the source nodes for the edges in the slice. A formal definition of a slice follows.

3.4.2 Definition 2: Slice of a PSDL Prototype: A *slice* $S_P(X)$ of a PSDL prototype P with respect to a set of data streams X is the subgraph (V, E, C) of the PDG G_P where:

(1) V is the smallest set that contains all vertices $o_i \in G_P$ that satisfy at least one of the following conditions:

   a) o_i writes to one of the data streams in X

   b) o_i precedes o_j in G_P and $o_j \in V$

(2) E is the smallest set that contains all of the edges $x_k \in G_P$, which satisfy at least one of the following conditions:

   a) $x_k \in X$

   b) x_k is directed to some $o_i \in V$

(3) C is the smallest set that contains all of the timing and control constraints associated with each operator in V and each data stream in E.

Figure 1 shows a prototype for a fish farm control system called *Fishies*. Figures 2, 3, and 4 display different slices of *Fishies*.

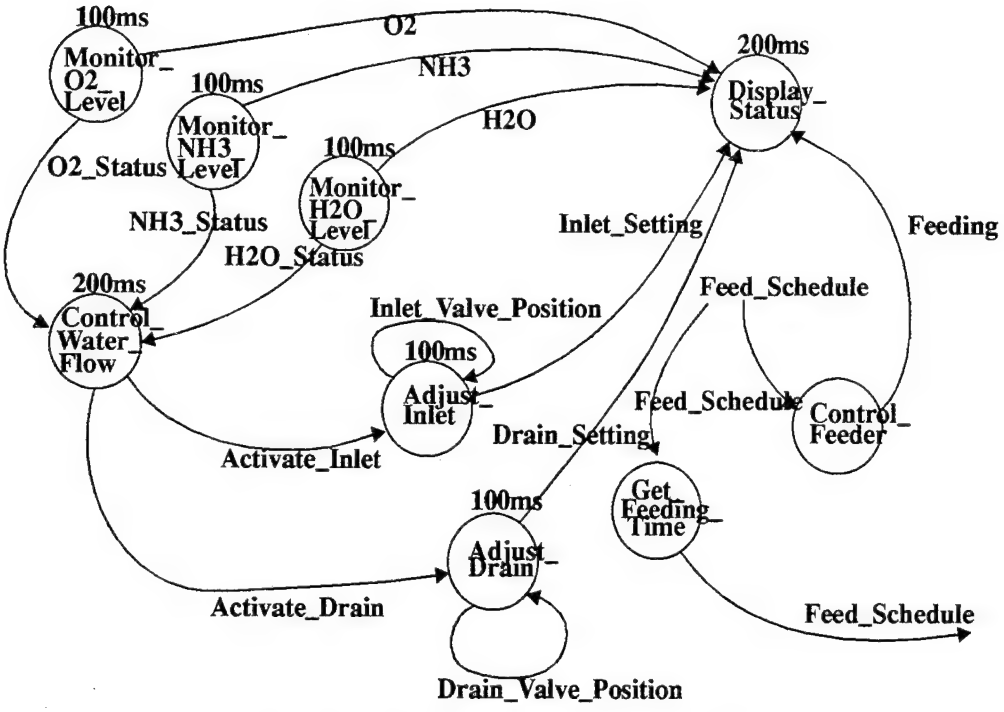


Figure 1. Fish farm control system, $fishies_{1.1}$.

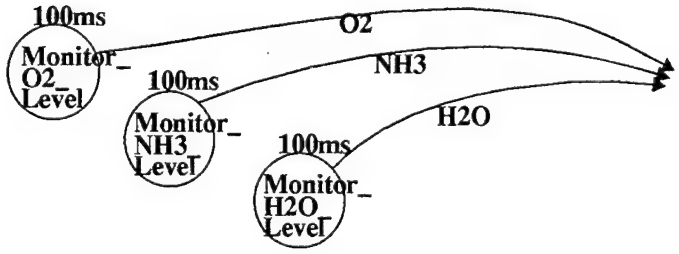


Figure 2. Slice of $fishies_{1.1}$ with respect to O2, NH3, H2O.

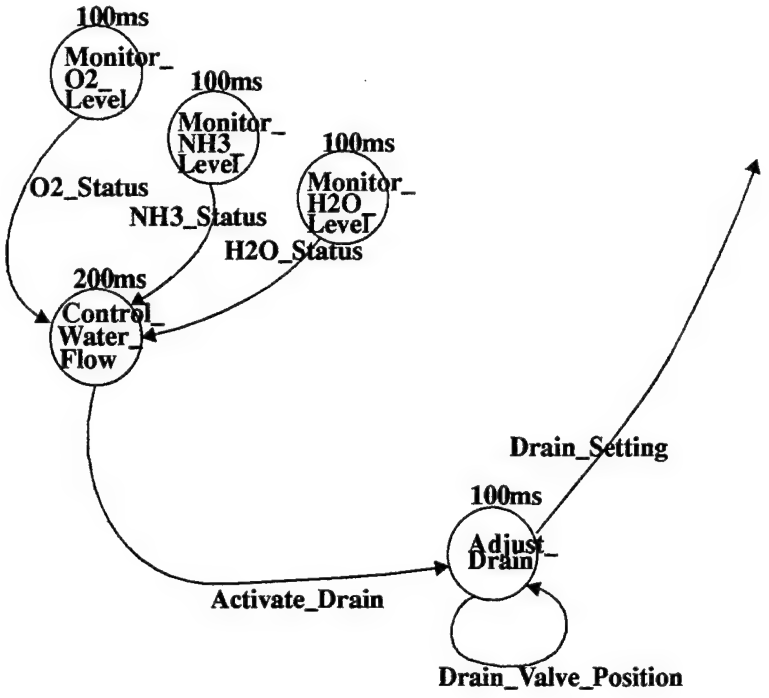


Figure 3. Slice of $fishies_{1,1}$ with respect to drain_setting.

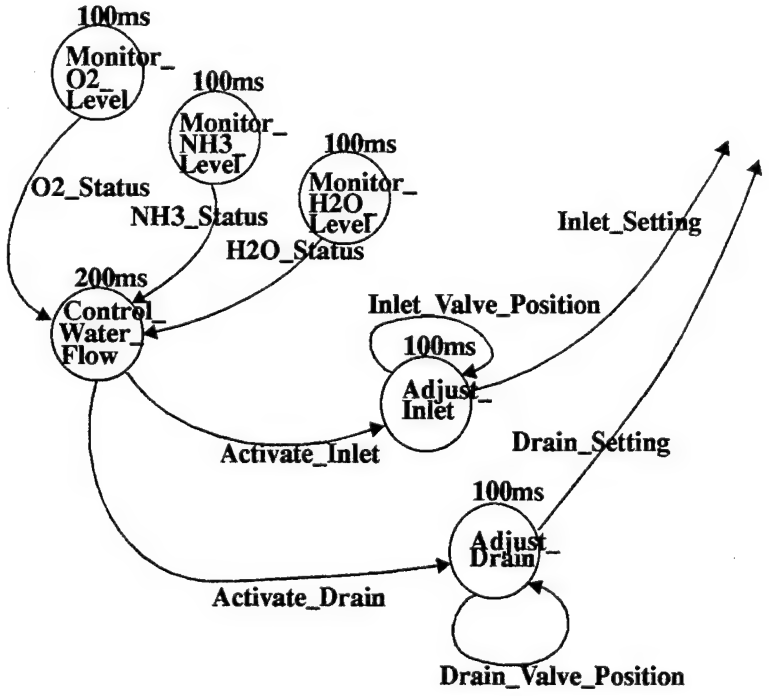


Figure 4. Slice of $fishies_{1,1}$ with respect to drain_setting and inlet setting.

3.4.3 Theorem: Slicing Theorem for PSDL Prototypes. Let $S_P(X)$ be the slice of a prototype P with respect to a set of streams X. Then $S_P(X)$ and P have the same behavior on any subset of the streams in $S_P(X)$.

The proof of this theorem is contained in (Dampier 1994). The significance of this theorem is that a slice captures a fragment of the semantic behavior of a prototype, and the behavior captured by that slice remains the same even if that slice is made a part of a different prototype, provided that it is also a slice with respect to that new prototype. This property is the basis for constructing a change-merging operation that can provide semantic guarantees of correctness.

4. METHOD

Our change-merging method for PSDL prototypes uses prototype slicing to determine automatically what parts of the prototype have been affected by a change and what parts have been preserved. Figures 5 and 6 show multiple modified versions of the *Fishies* prototype.

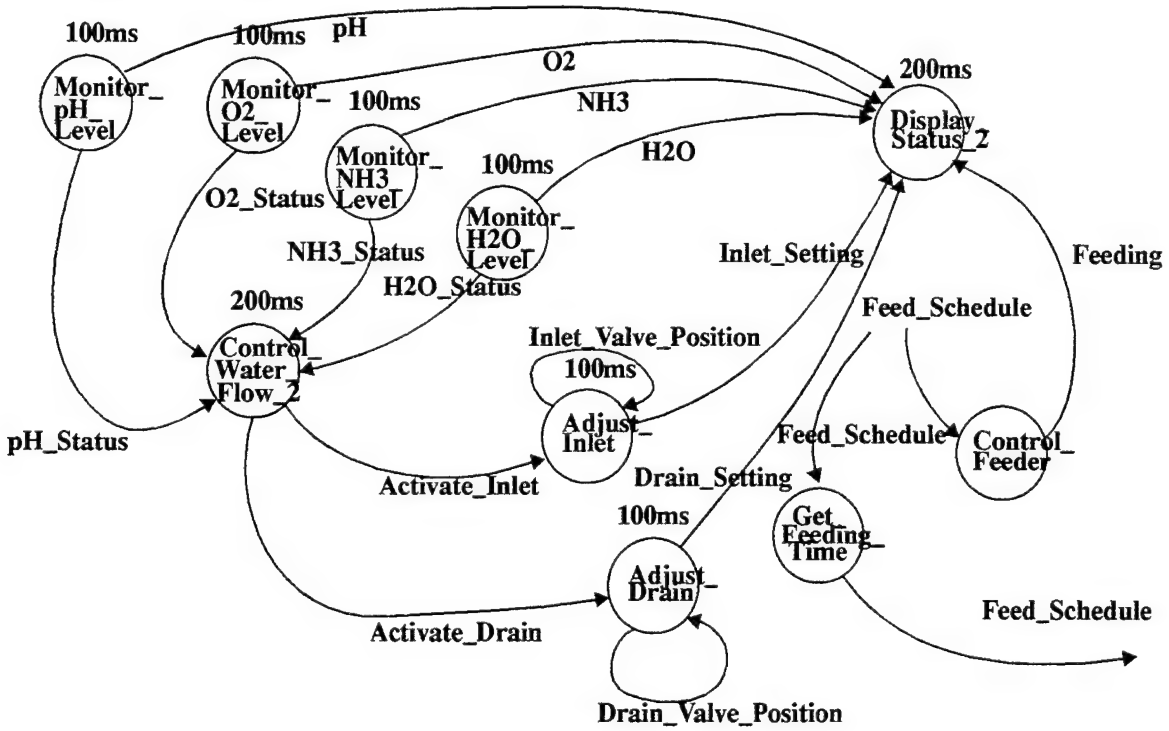


Figure 5. Version 1.2 of fish farm control system, $fishies_{1.2}$.

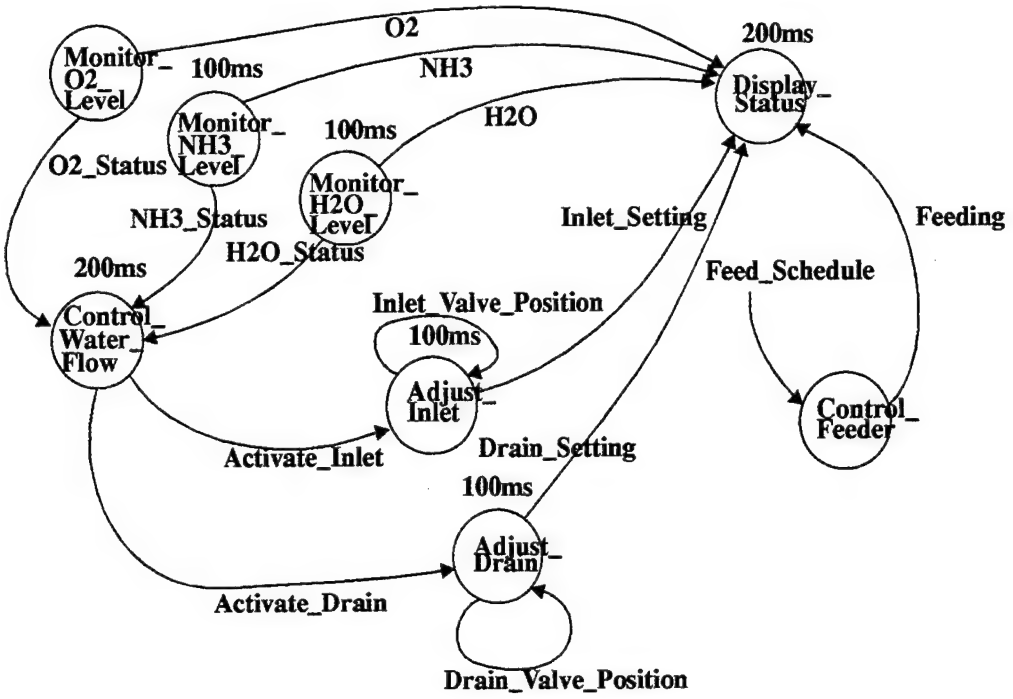


Figure 6. Version 2.2 of fish farm control system, *fishies*$_{2.2}$.

If the slice of a changed version of a prototype, with respect to a stream present in both the base version and the modified version, is different than the same slice of the base version, then the behavior on that slice is likely to be different. Therefore that change is significant, and must be preserved in the merged version. For example, consider the slice of *Fishies*$_{1.1}$ with respect to the stream *Activate_Drain* illustrated in Figure 7, and the same slice of *Fishies*$_{1.2}$, illustrated in Figure 8. It is easy to see a portion of the effect of the change that produced *Fishies*$_{1.2}$ from *Fishies*$_{1.1}$. If we were to take the same slice of *Fishies*$_{2.2}$, we would discover that it is identical to the slice of the base version *Fishies*$_{1.1}$. This illustrates that this part of the *Fishies* prototype is not affected by the change that produced *Fishies*$_{2.2}$. Since this change is significant in version 1.2, it must be reflected in the merged version.

Slices are important because they capture all of the parts of a program that can affect the behavior visible in a set of data streams. If two different programs have the same slice for a set of streams, they also have the same behavior over that set of streams. The preserved part of a prototype is then the largest set of streams that have the same single stream slice in all three versions, and the affected streams of each modification are those that have a different single stream slice in the modified version than in the base version. Performing a change-merge using *Fishies*$_{1.1}$ as the base version, and *Fishies*$_{1.2}$ and *Fishies*$_{2.2}$ as the modified versions, we get the preserved part as shown in Figure 9 and affected parts as shown in Figures 10 and 11.

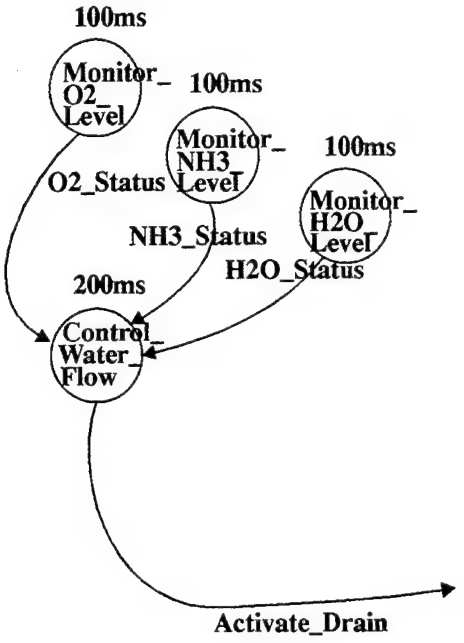


Figure 7. $S_{fishies1.1}$ (Activate_Drain).

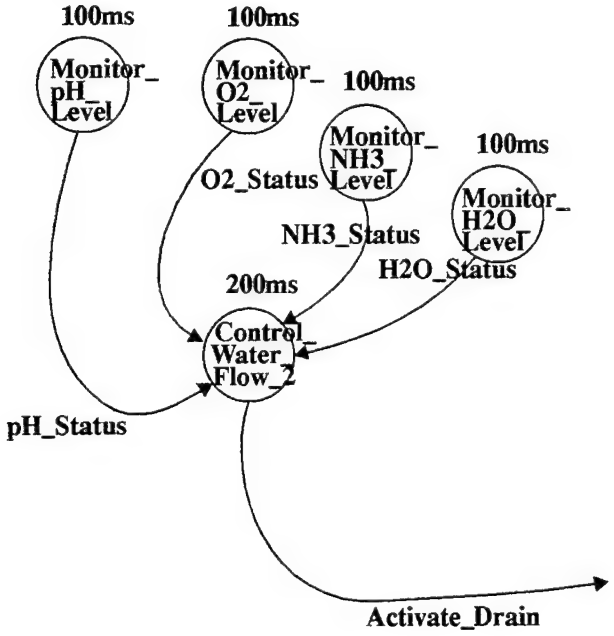


Figure 8. $S_{fishies1.2}$ (Activate_Drain).

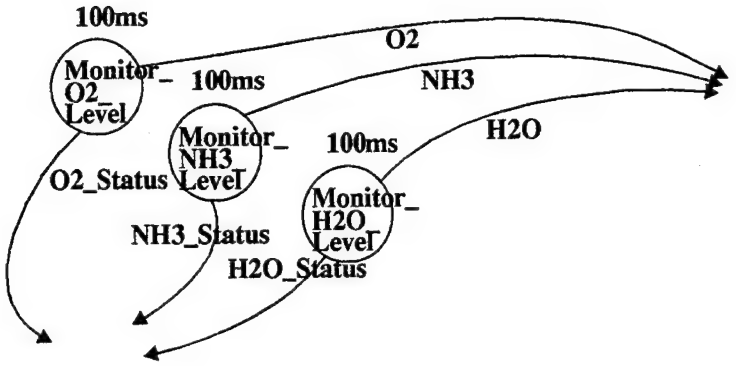


Figure 9. Preserved parts of *fishies*$_{1.1}$ in both modifications.

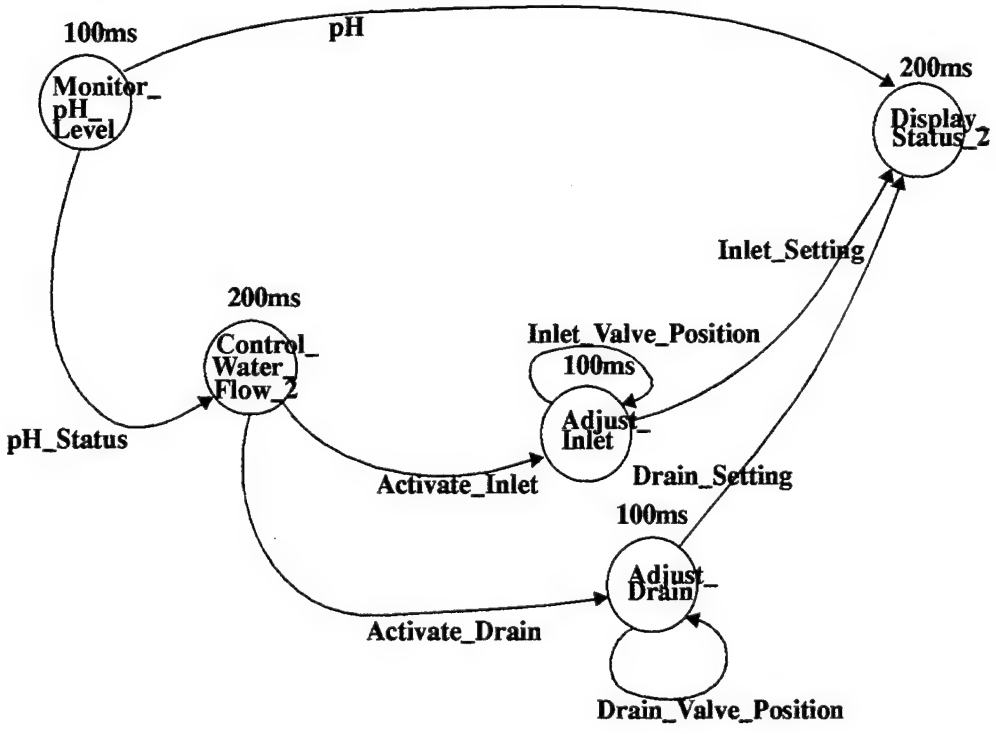


Figure 10. Affected part of *fishies*$_{1.2}$.

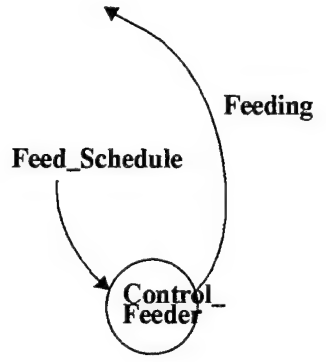


Figure 11. Affected part of *fishies*$_{2.2}$.

In constructing the preserved part, we consider each stream individually, taking the slice of each version with respect to that stream. If the slices are the same, then that slice is added to the preserved part. After all streams have been checked, the preserved part is complete.

The affected parts are constructed by comparing the slices of each stream in the modified version against the same slice of the base version. The stream is included in the affected part if the slices are different.

The merged version is formed by taking the union of the preserved part of all three versions and the affected parts of the two modified versions. If the slice of the merged version with respect to the streams affected by each modification is the same as the corresponding slice of the modified version, then semantic correctness of the merged version with respect to the modifications is established. The result of change-merging *Fishies*$_{1.1}$, *Fishies*$_{1.2}$, and *Fishies*$_{2.2}$ is shown in Figure 12.

Our slicing method has the advantage of a clear semantic criterion for correctness, and the disadvantage of reporting conflicts whenever two changes can affect the same stream, regardless of whether there exists a computation history in which the two changes actually interact or conflict with each other.

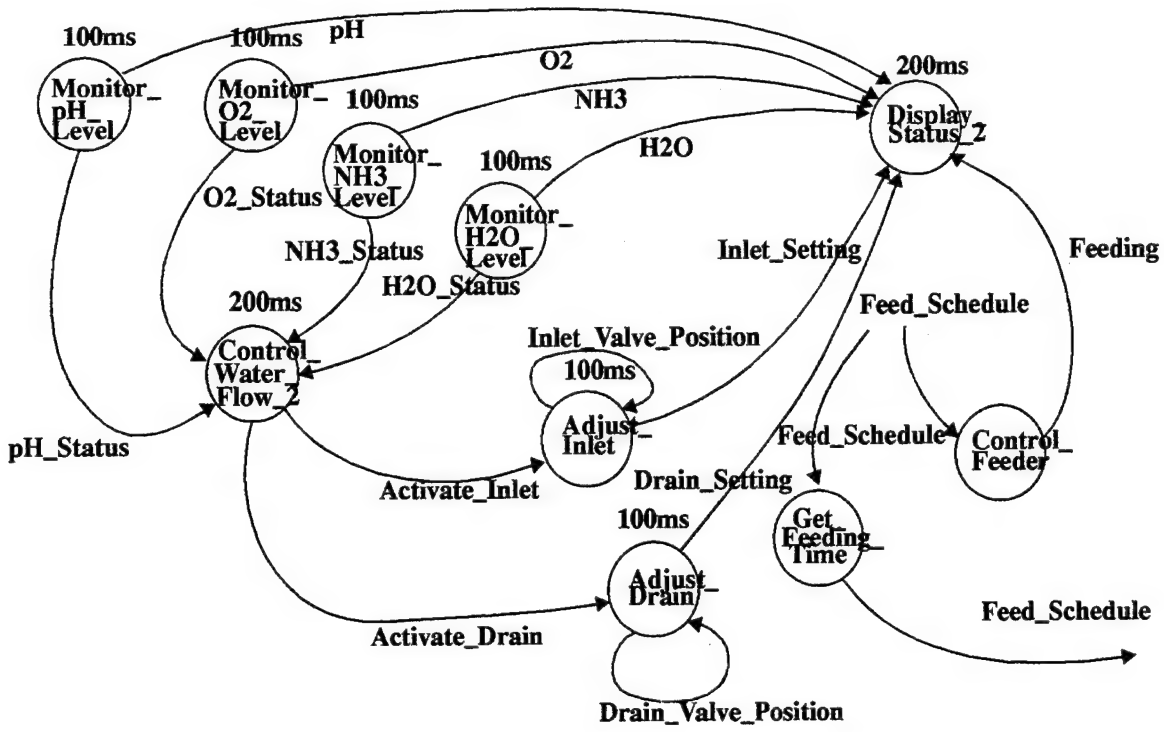


Figure 12. Change-merged version of the *fishies* prototype.

5. CHANGE-MERGE ALGORITHM

An algorithm for our method is shown in Figure 13. The sub-algorithms for each of the individual parts of *change_merge* can be found in Dampier (1994).

The algorithm *change_merge* accepts three expanded versions of a PSDL program as input. It then extracts all of the PSDL components from each version of the program. The atomic components are held in storage to be included in the change-merged version of the program, if needed. The composite component of each program is divided into a specification part and an implementation part.

Each of these parts are change-merged separately and the results are recombined to create the change-merged composite component. From the implementation part of the change-merged composite component, the algorithm can deduce which of the atomic components need to be included in the change-merged program. The change-merged program is then returned. If a conflict is detected during the change-merging process, the *CONFLICT* variable is set to true, and a flag is placed into the change-merged program at the location of the conflict to aid the designer in locating and resolving it.

Algorithm *change_merge*(BASE,A,B: in psdl_program;
CONFLICT: out boolean) return psdl_program

begin
1. Extract the psdl_components from each of the input programs.
2. Change-merge the specification parts for the three input composite components.
   a. Change-merge the state declarations.
   b. Change-merge the exception declarations.
   c. Change-merge the maximum execution times.
   d. Change-merge the formal and informal descriptions.
3. Change-merge the implementation parts for the three input composite components.
   a. Create the prototype dependency graphs for each version.
   b. Create the affected parts of each modified version.
   c. Create the preserved part of the base in all three versions.
   d. Change-merge the graphs.
   e. Change-merge the stream declarations.
   f. Change-merge the timer declarations.
   g. Change-merge the control constraints.
      (1) Change-merge the trigger constraints.
      (2) Change-merge the execution guard constraints.
      (3) Change-merge the periods.
      (4) Change-merge the finish_withins.
      (5) Change-merge the minimum calling periods.
      (6) Change-merge the maximum response times.
      (7) Change-merge the output guard constraints.
      (8) Change-merge the exception trigger constraints.
      (9) Change-merge the timer operations.
4. Create the change-merged program.
   a. Combine the change-merged specification and implementation.
   b. From the resulting implementation, determine which of the atomic components from each of the input versions is to be included in the change-merged program.
5. Return the change-merged program.

end *change_merge*;

Figure 13. Algorithm *change_merge*.

6. SUMMARY

We have provided a method for aiding the prototype designer in independently developing different parts of a software prototype, and automatically integrating the results of the independent efforts. This will allow multiple designers to work on the same prototype independently, or different versions of the prototype to be developed independently, with the knowledge that these independent results can be integrated after the fact with some guarantee of correctness.

Our method has been implemented and a tool is available for use with the CAPS prototyping environment to perform change-merging on real software prototypes. Research continues to provide more useful conflict resolution techniques and add the ability to change-merge abstract data types. Other future work will concentrate on realizing a method for change-merging programs written in an implementation language such as Ada.

INTENTIONALLY LEFT BLANK.

7. REFERENCES

Berzins, V. "On Merging Software Extensions." Acta Informatica, Springer-Verlag, pp. 607–619, 1986.

Dampier, D. "A Formal Method for Semantics-Based Change-Merging of Software Prototypes." Ph.D. Dissertation, Naval Postgraduate School, Monterey, CA, June 1994.

Horwitz, S., J. Prins, and T. Reps. "Integrating Non-Interfering Versions of Programs." Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, New York, NY, 13–15 January 1988.

Luqi, V. Berzins, and R. Yeh. "A Prototyping Language for Real Time Software." IEEE Transactions on Software Engineering, pp. 1409–1423, October 1988.

Reps, T. "On the Algebraic Properties of Program Integration." University of Wisconsin-Madison Technical Report CS-856, June 1989.

Reps, T., and W. Yang. "The Semantics of Program Slicing." University of Wisconsin-Madison Technical Report CS-777, 1988.

Silverberg, I. Source File Management with SCCS, Englewood Cliffs, NJ: Prentice-Hall, 1992.

Tichy, W. "Design, Implementation, and Evaluation of a Revision Control System." Proceedings of the 6th International Conference on Software Engineering, IEEE, Tokyo, pp. 58–67, September 1982.

Weiser, M. "Program Slicing." IEEE Transactions on Software Engineering, IEEE, pp. 352–357, July 1984.

Yang, W. "A New Algorithm for Semantics-Based Program Integration." University of Wisconsin-Madison Technical Report CS-962, 1990.

INTENTIONALLY LEFT BLANK.

NO. OF COPIES	ORGANIZATION
2	ADMINISTRATOR ATTN DTIC DDA DEFENSE TECHNICAL INFO CTR CAMERON STATION ALEXANDRIA VA 22304-6145
1	DIRECTOR ATTN AMSRL OP SD TA US ARMY RESEARCH LAB 2800 POWDER MILL RD ADELPHI MD 20783-1145
3	DIRECTOR ATTN AMSRL OP SD TL US ARMY RESEARCH LAB 2800 POWDER MILL RD ADELPHI MD 20783-1145
1	DIRECTOR ATTN AMSRL OP SD TP US ARMY RESEARCH LAB 2800 POWDER MILL RD ADELPHI MD 20783-1145
	ABERDEEN PROVING GROUND
5	DIR USARL ATTN AMSRL OP AP L (305)

NO. OF COPIES	ORGANIZATION
1	DIR USARL ATTN AMSRL SC I 115 OKEEFE BLDG ATLANTA GA 30332-0800
50	DIR USARL ATTN AMSRL SC IS 115 OKEEFE BLDG ATLANTA GA 30332-0800
2	DIR USARL ATTN AMSRL SC IS CPT DAVID A DAMPIER 115 OKEEFE BLDG ATLANTA GA 30332-0800
2	NAVAL POSTGRADUATE SCHOOL ATTN DR VALDIS BERZINS COMPUTER SCIENCE DEPT 833 DYER RD MONTEREY CA 93943
2	NAVAL POSTGRADUATE SCHOOL ATTN DR LUQI COMPUTER SCIENCE DEPT 833 DYER RD MONTEREY CA 93943
2	NAVAL POSTGRADUATE SCHOOL ATTN DR MANTAK SHING COMPUTER SCIENCE DEPT 833 DYER RD MONTEREY CA 93943
1	NAVAL POSTGRADUATE SCHOOL ATTN DR DAN DOLK SYSTEMS MANAGEMENT DEPT MONTEREY CA 93943
1	NAVAL POSTGRADUATE SCHOOL ATTN DR CRAIG RASMUSSEN MATHEMATICS DEPT MONTEREY CA 93943
	ABERDEEN PROVING GROUND
1	DIR USARL ATTN AMSRL SC

USER EVALUATION SHEET/CHANGE OF ADDRESS

This Laboratory undertakes a continuing effort to improve the quality of the reports it publishes. Your comments/answers to the items/questions below will aid us in our efforts.

1. ARL Report Number ARL-TR-840 Date of Report August 1995

2. Date Report Received _____

3. Does this report satisfy a need? (Comment on purpose, related project, or other area of interest for which the report will be used.) _____

_____

_____

4. Specifically, how is the report being used? (Information source, design data, procedure, source of ideas, etc.)

_____

_____

_____

5. Has the information in this report led to any quantitative savings as far as man-hours or dollars saved, operating costs avoided, or efficiencies achieved, etc? If so, please elaborate. _____

_____

_____

6. General Comments. What do you think should be changed to improve future reports? (Indicate changes to organization, technical content, format, etc.) _____

_____

_____

_____

CURRENT ADDRESS

_____
Organization

_____
Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

7. If indicating a Change of Address or Address Correction, please provide the Current or Correct address above and the Old or Incorrect address below.

OLD ADDRESS

_____
Organization

_____
Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

(Remove this sheet, fold as indicated, tape closed, and mail.)

(DO NOT STAPLE)